Old Dominion University Research Foundation

NAA/LANGLEY
NCC 1-99
IN-61
64848-CR
P.42

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23508

GEOMETRIC MODELING FOR COMPUTER-AIDED DESIGN

By

James L. Schwing, Principal Investigator

and

Jan Spangler, Graduate Research Assistant

Progress Report
For the period ended December 31, 1986

Prepared for the
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Under
Research Grant NCC1-99
John J. Rehder, Technical Monitor
SSD-Vehicle Analysis Branch

(NASA-CR-180686) GEOMETRIC MODELING FOR COMPUTER-AIDED DESIGN Progress Report, period ending 31 Dec. 1986 (Old Dominion Univ.) 42 p Avail: NTIS HC A03/MF A01 CSCL 09B G3/61

N87-27423

Unclas
0064848

March 1987

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23508

GEOMETRIC MODELING FOR COMPUTER-AIDED DESIGN

By

James L. Schwing, Principal Investigator

and

Jan Spangler, Graduate Research Assistant

Progress Report
For the period ended December 31, 1986

Prepared for the
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Under
Research Grant NCC1-99
John J. Rehder, Technical Monitor
SSD-Vehicle Analysis Branch

Submitted by the
Old Dominion University Research Foundation
P. O. Box 6369
Norfolk, Virginia 23508

March 1987

TABLE OF CONTENTS

I. Introduction

The following report summarizes the research carried out during the recently completed phase of NASA grant NCCI-99. The work described here was carried out by the principal investigator, James Schwing, and a graduate research assistant, Jan Spangler, in conjunction with the SMART system design team (Solid Modeling Aerospace Research Tool) of the Vehicle Analysis Branch at NASA Langley.

The major effort of the past six months has been the development of software used with the derivation of smooth 3-D surfaces from a sequence of cross-sections. Additional work has considered on problems arising in the creation of surfaces by extrusion and the presentation of calculated physical properties.

II. Minimizing Error in Surface Representation.

A. The Problem.

The basis for geometric representation used in the SMART system is the bicubic parameterization known as the Bezier patch. Refer to Foley and Van Dam [1] for example for a discussion of basic patch definition and manipulation. The SMART system attempts to provide interfaces for the design engineer that corresponds to natural engineering design and development tools. Thus one of the geometric input techniques provided allows the designer to input a sequence of cross-sections of the object under consideration. The problem then becomes one of converting this sequence of cross-sections to a collection of Bezier patches that reproduces the given data as accurately as possible.

The key to this conversion lies in the calculation of Bezier curves, edges of the Bezier patches, which approximate the cross-sections in a way that minimizes error. A straight forward solution to this would seem to be the calculation of these Bezier curves via a least squares technique. However, the process is complicated by the fact that the calculated Bezier curves are then used as the edges of the Bezier patches that are expected to join together in a smooth, differentiable fashion. This imposes additional interaction conditions on the calculation of the least square Bezier curves.

Description of these additional conditions can best be seen by considering what happens when two patches are to be joined in a differentiable fashion. The reader is again referred to Foley and Van Dam [1] for more detail. To summarize let Patch 1 be represented by the 16 control points $P_{11} \ldots P_{44}$ and Patch 2 be represented by $Q_{11} \ldots Q_{44}$. The conditions can now be described in terms of these control points. Suppose that the common edge is given by the control points P_{14}, P_{24}, P_{34}, P_{44} in Patch 1 and by the control points Q_{11}, Q_{21}, Q_{31}, Q_{41} in Patch 2. The first requirement is that $P_{i4} = Q_{i1}$ for i = 1, 2, 3, and 4. In addition, the following relation must hold for the interior control points: $Q_{i2} = Q_{i1} + \mathbf{k} (Q_{i1} - P_{i3})$ with **k** constant and i = 1, 2, 3, and 4.

As an example let the two curves shown below in figure 1 represent a portion of two consecutive cross-sections where each portion is represented by

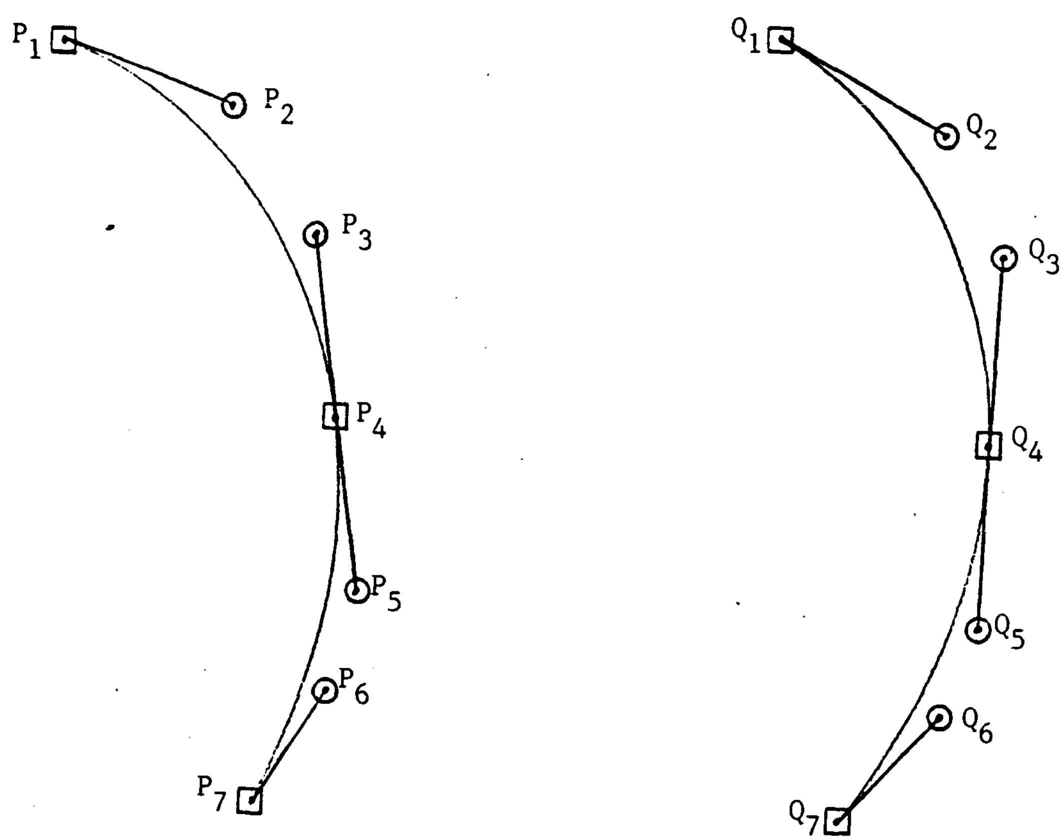

Figure 1

the joining of two Bezier curve segments. The condition cited above requires that

$$P_5 = P_4 + k\,(P_4 - P_3) \qquad \text{and}$$

$$Q_5 = Q_4 + k\,(Q_4 - Q_3) \qquad \text{for some } k.$$

Since each cross-section is adjacent to at least one other cross-section the value of **k** must be determined for all cross-sections simultaneously.

B. The Theory.

Assume that the Bezier curve segments which approximate a given cross-section are ordered and that the Bezier control points for the previous segments have already been determined. That is, in each cross-section there is a previously determined Bezier curve segment, such as $P_1, \ldots, P_4$, to which a new Bezier curve segment, such as $P_4, \ldots, P_7$, will be joined. The special case of curve segments where there is no previous information will be treated separately.

Figure 2 below represents the segment to be calculated taken from say the jth cross-section. The given data points represent points of the jth cross-section. Interpolation conditions for the Bezier curve require that the first and fourth control points of this portion correspond to the first and last data points. Thus the values for P_{0j} and P_{1j} are easily determined. By assumption stated above the control points of the previous segment are known, specifically c_{1j}. The requirements noted in the previous section state that the control point C_{0j} must satisfy with constant $\mathbf{k}$:

$$C_{0j} = P_{0j} + \mathbf{k}\,(P_{0j} - c_{1j}) \qquad \text{for all cross-sections } \mathbf{j}.$$

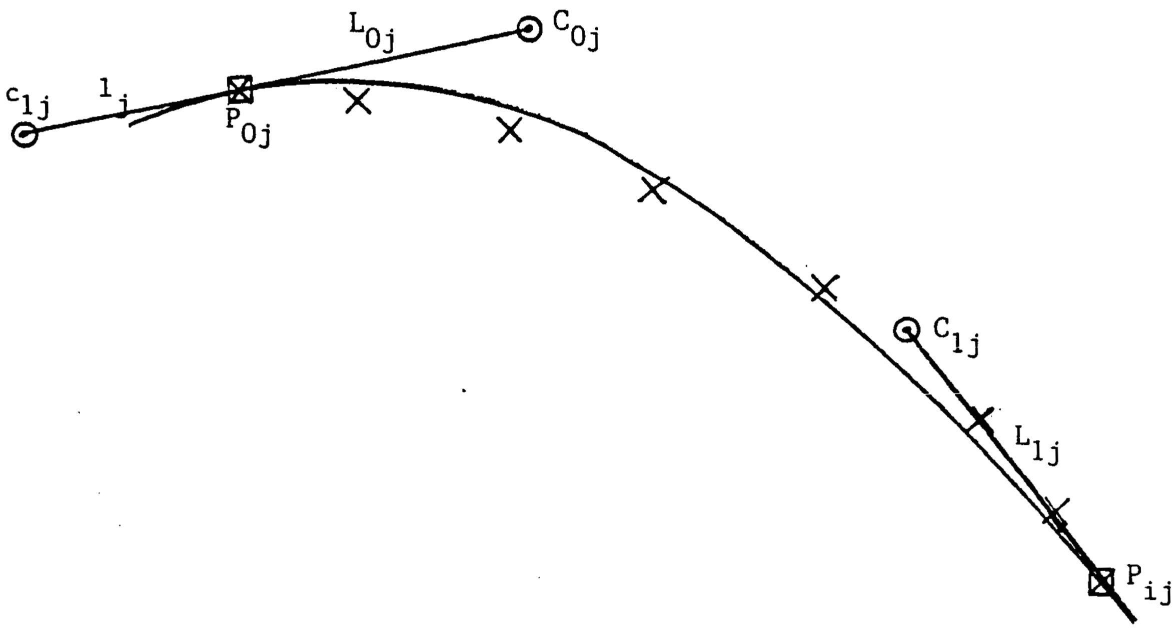

Figure 2

The least squares technique is now applied to find those values of $\mathbf{k}$ and C_{1j} that minimize the error made by approximating the data points by the Bezier curves. To solve the least squares problem this portion of the curve is parameterized by chord length along the cross-section data points. Let those distances be represented by s_{ij}. Further let l_j, L_{0j} and L_{1j} represent the distances and $\mathbf{d}_j$ and $\mathbf{D}_j$ represent the direction vectors necessary to define the control points as follows:

$$c_{1j} = P_{0j} - l_j \, \mathbf{d}_j$$

$$C_{0j} = P_{0j} + L_{0j} \, \mathbf{d}_j$$

$$C_{1j} = P_{1j} + L_{1j} \, \mathbf{D}_j$$

and where it is known that $L_{0j} = \mathbf{k} \, l_j$.

The least square solution then leads to both of the following types of equations.

Eqn. 1: $$a_j \, \mathbf{k} + b_j \, L_{1j} = c_j$$

Eqn. 2: $$\sum_{j=1}^{N} [d_j \, \mathbf{k} + e_j \, L_{1j}] = \sum_{j=1}^{N} f_j$$

where

$$a_j = 3 \, l_j \, [\sum_{i=1}^{n_j-1} (1 - s_{ij})^3 \, s_{ij}^3] \, [d_{x0j} \, d_{x1j} + d_{y0j} \, d_{y1j}]$$

$$b_j = 3 \, [\sum_{i=1}^{n_j-1} (1 - s_{ij})^2 \, s_{ij}^4] \, [d_{x1j}^2 + d_{y1j}^2]$$

$$c_j = \sum_{i=1}^{n_j-1} \{(1 - s_{ij}) \, s_{ij}^2 \, [x_{ij} \, d_{x1j} + y_{ij} \, d_{y1j}]\}$$

$$- \{\sum_{i=1}^{n_j-1} (1 - s_{ij})^3 \, s_{ij}^2 \, (1 + 2 \, s_{ij})\} [x_{0j} \, d_{x1j} + y_{0j} \, d_{y1j}]$$

$$- \{\sum_{i=1}^{n_j-1} (1 - s_{ij}) \, s_{ij}^4 \, (3 - 2 \, s_{ij})\} [x_{n_j j} \, d_{x1j} + y_{n_j j} \, d_{y1j}]$$

$$d_j = 3 \, l_j^2 \, [\sum_{i=1}^{n_j-1} (1 - s_{ij})^4 \, s_{ij}^2] [d_{x0j}^2 + d_{y0j}^2]$$

$$e_j = a_j$$

$$f_j = l_j \{ \sum_{i=1}^{n_j-1} (1 - s_{ij})^2 s_{ij} [x_{ij} d_{x0j} + y_{ij} d_{y0j}]$$

$$-[\sum_{i=1}^{n_j-1} (1 - s_{ij})^4 s_{ij} (1 + 2 s_{ij})][x_{0j} d_{x0j} + y_{0j} d_{y0j}]$$

$$-[\sum_{i=1}^{n_j-1} (1 - s_{ij})^2 s_{ij}^3 (3 - 2 s_{ij})] [x_{n_j j} d_{x0j} + y_{n_j j} d_{y0j}] \}$$

Eqn.s 1 and 2 can be solved for all j by the following:

$$k = \sum_{i=1}^{N}[f_j - (a_j \cdot c_j)/b_j] \; / \; \sum_{i=1}^{N}[d_j - a_j^2/b_j]$$

$$L_{1j} = [c_j - (a_j \cdot k)] \; / \; b_j \; .$$

For the special case that considers the first segment of each cross-section, the interaction condition weakens. This is a consequence of the fact that these patches will have no continuity condition on their leading edge. Thus the ratio constraint previously described no longer applies and the values for L may be calculated at both ends of the curve.

Fortunately, the solution of the resulting least squares problem for this case leads to virtually the same coefficients, a_j, b_j, c_j, d_j, e_j, f_j as listed above. One need only set l_j to 1. The solution to the equations in this special case is given by:

$$L_{0j} = [b_j \cdot f_j - a_j \cdot c_j] \; / \; [b_j \cdot d_j - a_j^2]$$

$$L_{1j} = [c_j \cdot d_j - a_j \cdot f_j] \; / \; [b_j \cdot d_j - a_j^2] \; .$$

C. The Results.

As implemented these routines will take as input a collection of cross-sections and produce a best least squares approximant satisfying the conditions necessary for the building of a smooth Bezier surface. The appendix includes the code used for implementing these ideas.

III. Solids Via Extrusion.

Here the basic idea is that a given solid may be defined by the act of dragging a fixed, user-defined cross-section along another user-defined path. No major restrictions are placed upon either the cross-section, which may be any planar curve, or the path, which may be any 3-D curve. Both are represented internally by the standard SMART format of connected Bezier curve segments. As in the previous section these Bezier curves are to be used to generate the corresponding surface patches of the solid being defined. This research addressed the problem of providing the same continuity and shape in the resulting solid as that of the underlying extrusion path.

There are two aspects to the problem mentioned above. First, when segments of the underlying extrusion curve join in a continuous fashion, it is necessary to join the resulting surfaces in a continuous fashion. Secondly, the shape of the "tube" generated by the extrusion should accurately reflect the user-defined cross-section. The solution to the first half of this problem is discussed in the latter part of this chapter.

With respect to the second half of the problem, the solution is not immediately obvious. Unfortunately, the process is not "well defined." That is, it is not possible to completely determine all of the final parameters from the two curves described above as the user input. In effect, there is one remaining parameter left to be freely picked by the software.

To this point all existing automatic techniques that have been employed to determine this final parameter lead to an undesirable twist in the extruded surface in some cases. That is, the resulting surface appears to twist so that the inside is totally constricted. Figure 3 below illustrates this condition. Currently we are still working on a solution to this problem.

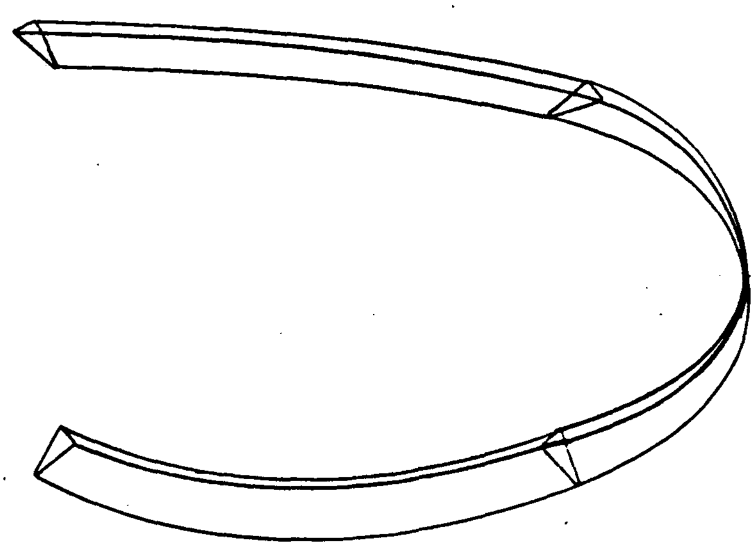

Figure 3

The key to the first half of the problem again rests in the proper determination of the constant ratio **k** mentioned in the previous section. The solution proposed below not only will keep the continuity of the joining surfaces the same as that of the underlying extrusion curve, it also attempts to reproduce as faithfully as possible the shape of the underlying extrusion curve. In order to do that the following aesthetic was adopted:

Aesthetic: If the underlying extrusion curve is either linear or circular reproduce the result exactly.

Note first that Bezier patch control points are known at the join, since they are given precisely by the user-defined cross-section placed at the join and oriented so that it is normal to the extrusion curve. Thus the solution to this problem reduces to identifying the appropriate values of the patch control points immediately preceding and following this join.

Let the extrusion curve preceding the join have the control points C_1, C_2, C_3, C_4. Similarly let C_4, C_5, C_6, C_7 represent the control points of the curve segment following the join. Define M_1 and M_2 as the "midpoints" of these respective curve segments.

$$M_1 = (C_1 + 3\ C_2 + 3\ C_3 + C_4)\ /\ 8$$

$$M_2 = (C_4 + 3\ C_5 + 3\ C_6 + C_7)\ /\ 8$$

Finally define W to be the center of the circle containing the three points M_1, M_2, C_4. Note that if these point are collinear, then W cannot be defined but that we have the simple case of reproducing the new control points in a linear fashion. If on the other hand the points are not collinear, then the center W is used to compute an appropriate radius of curvature for each of the given control points, P, in the user-defined cross-section. This radius is then used to produce the patch control points preceding and following the join the correspond to P. Specifically, we determine the previously mentioned constant **k** from the underlying curve as follows:

Eqn. 3 $$\mathbf{k} = \| \overline{C_3 C_4} \| / \| \overline{C_4 C_5} \|.$$

By using this **k** in the determination of all interior preceding and following Bezier points, the appropriate continuity class is assured.

Let P_p and P_f be the interior control points that precede and follow P in the definition of the Bezier surfaces meeting at the join. Notice that the following relations must hold:

Eqn. 4 $$P_p = P + r_1 (C_3 - C_4)$$

Eqn. 5 $$P_f = P + r_2 (C_5 - C_4).$$

As mentioned above, it is desired for continuity sake that these points share the common **k** value, that is:

$$\mathbf{k} = \| \overline{P P_p} \| / \| \overline{P P_f} \|.$$

The definitions of P_p and P_f above show that

$$\begin{aligned} \mathbf{k} &= r_1 \| \overline{C_3 C_4} \| / r_2 \| \overline{C_4 C_5} \| \\ &= (r_1 / r_2) (\| \overline{C_3 C_4} \| / \| \overline{C_4 C_5} \|) \\ &= (r_1 / r_2)\, \mathbf{k}. \end{aligned}$$

This implies that $r_1 = r_2$. At this point it is possible to combine this requirement and Eqn.s 3, 4, and 5 with the previously mention aesthetic to derive values for P_p and P_f. Basically the center of curvature W is used with these facts via:

- Linear Case: $r_1 = r_2 = 1$

  Non-linear Case: $r_1 = r_2 = \| \overline{P W} \| / \| \overline{C_4 W} \|$.

Equations for the calculation of W are straight forward in both two and three dimensions and can be found in any calculus text.

IV. Display of Physical Properties.

Calculation of physical properties is an important step in the analysis of aerospace vehicles. During the process of conceptual design such calculations must be relatively efficient without sacrificing significant accuracy so that a multiplicity of ideas can be tried rapidly with a reasonable confidence in the results. Previous research under this grant produced mathematical software provided accuracy beyond the required tolerances and which proved to be up to four times faster. Work over the last six months on this topic involved developing an appropriate user interface for the display of this information.

It has turned out that it is natural to represent the geometry for aerospace systems in tree data structures. These trees capture the hierarchical relation between system components and their subassemblies. Since the physical properties are calculated for each of the basic subassemblies and then propagated through the hierarchy to the more complex components. This information combined with the fact that the designer is already interacting with this hierarchy through the mouse dictated the following design of the interface.

Once the designer selects the calculation of physical properties, a dual viewport display is presented. One of the viewports contains a representation of the current hierarchy. From this viewport the user may use the mouse to move around the data structure and select a particular component or subassembly for which the properties should be presented. The selected information is then immediately displayed in the other viewport. In a sense this allows the designer to browse the model each time design changes have been made.

REFERENCE

1. Foley, J. and A. Van Dam, Principles of Interactive Computer Graphics, Addison Wesley, 1982.

APPENDIX

ROUTINES USED IN THE GENERATION OF MINIMUM ERROR BEZIER CURVES CORRESPONDING TO A SET OF CROSS SECTIONS.

INCLUDE FILE DEPENDENCIES

```
<sgimath.h>
../include.dir/act_data.h
```

DEFINED FUNCTONS AND VARIABLES

```
sqr(X)   (X) * (X)
MAXCPCSP1 11        /* represents MAXCPCS + 1 */
MAXCPCSP2 12        /* represents MAXCPCS + 2 */
```

ROUTINE: gen_bz

PURPOSE: driving routine for the generation of minimum error Bezier control points for Bezier curves approximating a set of cross sections

CALLING PROCEDURE/
  DECLARATIONS:

```
gen_bz(xp,yp,zp,ifl,ncs,npcs,cpts,err)

float xp[][MAXCS],yp[][MAXCS],zp[],*cpts[][MAXCS][4][3];
int   ifl[][MAXCS],ity,ncs,npcs[],*err;
```

INPUT VARIABLES:

xp,yp - [MAXCPCS][MAXCS], arrays containing the cross section curves
zp - [MAXCS], array containing the z position of the given cross section
ifl - [MAXPPCS][MAXCS], array containing an indicator

```
                         for curve segment break points and continuity
                         conditions
                    ncs - actual number of cross sections used
                    npcs - [MAXCS], array listing the actual number of points
                         used in each cross section

OUTPUT VARIABLES:
                    cpts - [MAXCPCS][MAXCS][4][2], array containing the x,y
                         coordinates of the 4 Bezier control points for each
                    err - if non-zero indicates an error exit condition
                         due to inconsistant data
                         curve and each cross section

EXTERNAL VARIABLES:    none

GLOBAL VARIABLES:      none

INTERNAL VARIABLES:
                    n - [MAXCPCSP2][MAXCS], array containing the actual numbe
                         of points in each curve segment
                    nseg - the actual number of curve segments for any cross
                         section
                    cs, segno - counters
                    brkpt - [MAXCPCSP1][MAXCS], array containing the index
                         to xp,yp for the break points between the Bezier
                         curve segments
                    dvect_in,dvect_out - [MAXCPCSP1][MAXCS][2], array containi
                         the x,y components of the incomming/outgoing
                         tangent directions at each break point
                    x,y - [MAXPPCS][MAXCS], array containing the coordinates c
                         a selected curve segment for each cross section
                    d0,d1 - [MAXCS][2], array containing the x,y components
                         for the start/end derivative direction on a given
                         curve segment over all cross sections
                    lambda - [MAXCS][[2], array containing the length of each
                         of hte control points from the start/end points
                         of a given curve segment over all cross sections
                    lambda_old - [MAXCS], array containing the length of the
                         end control point of the previous section stored
                         over all cross sections

ROUTINES INVOKED:
                    seg_info
                    get_dvect
                    sep_seg
                    bz_minerr1
                    get_cpts

AUTHOR:             James Schwing, Old Dominion University

DATE:               12-24-86
```

REVISIONS:

DATES:

ROUTINE: seg_info

PURPOSE: routine used to obtain information about the curve segments that will be used to approximate the cross sections

CALLING PROCEDURE/
DECLARATIONS:

```
seg_info(ifl,ncs,npcs,n,&nseg,brkpt,&err)

int ifl[][MAXCS],ncs,npcs[],n[][MAXCS],*nseg,brkpt[][MAXC
    *err;
```

INPUT VARIABLES:
ifl,ncs,npcs - as described in "gen_bz" above

OUTPUT VARIABLES:
n,nseg,brkpt,err - as described in "gen_bz" above

EXTERNAL VARIABLES: none

GLOBAL VARIABLES: none

INTERNAL VARIABLES:
cs,seg,i,j - counters
segcnt - track the number of curve segments from cross section to cross section

ROUTINES INVOKED: none

AUTHOR: James Schwing, Old Dominion University

DATE: 12-24-86

REVISIONS:

DATES:

ROUTINE: get_dvect

PURPOSE: routine used to produce incoming and outgoing tangent directions for each curve segment break point

CALLING PROCEDURE/

```
DECLARATIONS:
                    get_dvect(xp,yp,ncs,nseg,n,ifl,brkpt,dvect_in,dvect_out)

                    float xp[][MAXCS],yp[][MAXCS],
                          dvect_in[][MAXCS][2],dvect_out[][MAXCS][2];
                    int   ncs,nseg,n[][MAXCS],ifl[][MAXCS],brkpt[][MAXCS];

INPUT VARIABLES:
                    xp,yp,ncs,nseg,n,ifl,brkpt - as described in "gen_bz"

OUTPUT VARIABLES:
                    dvect_in,dvect_out - as described in "gen_bz"

EXTERNAL VARIABLES:       none

GLOBAL VARAIBLES:         none

INTERNAL VARIABLES:
                    a,b - [4], array containing tangent approximation info
                    ct,st - x,y components of the tangent vector
                    v - [2], temporay vector
                    len - vector length
                    xe1,xe2,ye1,ye2 - extrapolated points
                    cs,seg,i - counters
                    n1 - the number of points used to define the incoming
                         curve segment at a given break point
                    n2 - the number of points used to define the outgoing
                         curve segment at a given break point
                    ibase - index to xp,yp for the current break point

ROUTINES INVOKED:
                    tancalc
                    ln_vect
                    extrap
                    setscl

AUTHOR:             James Schwing, Old Dominion University

DATE:               12-24-86

REVISIONS:

DATES:

------------------------------------------------------------------------

ROUTINE:            bz_minerr1

PURPOSE:            routine used to calculate the length of the Bezier
                    control points along their tangent vectors so that
                    a minimum least square error for a given curve segment
                    over all cross sections between the approximating
```

```
                 Bezier curve and the cross section data is obtained

CALLING PROCEDURE/
  DECLARATIONS:
        bz_minerr1(x,y,d0,d1,ncs,n,segno,lambda_old,lambda);

        float x[][MAXCS],y[][MAXCS],d0[][2],d1[][2],lamda_old[],*lambda[][2]
        int   ncs,n[][MAXCS],segno;

INPUT VARIABLES:
                x,y,d0,d1,ncs,n,segno,lambda_old - as described in "gen_bz"

OUTPUT VARIABLES:
                lambda - as described in "gen_bz"

EXTERNAL VARIABLES:     none

GLOBAL VARIABLES:       none

INTERNAL VARIABLES:
                dist - chord length for this segment.
                s - array for chord length position of data defining
                        this segment.
                        the second index represents powers of s.
                scomp - (1 - s)
                        the second index represents powers.
                sx2m3 - (3 - 2 * s)
                sx2p1 - (2 * s + 1)
                l1, l2, l3 - accumulators for the LHS of the matrix
                        equation.
                r1, r2, r3, r4, r5, r6 - accumulators for the RHS
                        of te matrix equation.
                a, b, c, d, f - matrix entries used to find the
                        least squares solution.
                r - common ratio reqired for surface continuity,
                        found in the solution of the matrix system.
                tot1, tot2 - used in solution of the matrix system.
                n_cs - number of points for this segment and this
                        cross section.
                nm1 - (n_cs - 1)
                nm2 - (n_cs - 2)

ROUTINES INVOKED:
                sqr
                sqrt
                solve1
                solve2

AUTHOR:         James Schwing, Old Dominion University

DATE:           12-24-86
```

```
REVISIONS:

DATES:
------------------------------------------------------------------------

ROUTINE:          ln_vect

PURPOSE:          routine used to find the length of a 2-D vector

CALLING PROCEDURE/
  DECLARATIONS:
                  ln_vect(v)

                  float v[2];

INPUT VARIABLES:
                  v - [2], vector for which the length will be calculated

OUTPUT VARIABLES:
                  ln_vect - function value, the length

EXTERNAL VARIALBLES:      none

GLOBAL VARIABLES:         none

INTERNAL VARAIBLES:
                  len - temporary storage for length

ROUTINES INVOKED:
                  sqr
                  sqrt

AUTHOR:           James Schwing, Old Dominion University

DATE:             12-24-86

REVISIONS:

DATES:
------------------------------------------------------------------------

ROUTINE:          setscl

PURPOSE:          routine which modifies start/end tangent derivatives

CALLING PROCEDURE/
  DECLARATIONS:
                  setscl(&ct,&st)

                  float *ct,*st;
```

```
INPUT VARIABLES:         none

OUTPUT VARIABLES:
                    ct,st - x,y components of the tangent vector

EXTERNAL VARIABLES:      none

GLOBAL VARIABLES:        none

INTERNAL VARIABLES:      none

ROUTINES INVOKED:        none

AUTHOR:             James Schwing, Old Dominion University

DATE:               12-24-86

REVISIONS:

DATES:
---------------------------------------------------------------------

ROUTINE:            extrap

PURPOSE:            routine used to extrapolate points of a curve segment
                    beyond (or prior to) its end points

CALLING PROCEDURE/
  DECLARATIONS:
                    float extrap(a,b,c)

                    float a,b,c;

INPUT VARIABLES:
                    a,b,c - input coordinates from the curve segment; x or y

OUTPUT VARIABLES:
                    extrap - function value, projected coordinate value

EXTERNAL VARIABLES:      none

GLOBAL VARIABLES:        none

INTERNAL VARIABLES:
                    d - temporary storage

ROUTINES INVOKED:        none

AUTHOR:             James Schwing, Old Dominion University
```

```
DATE:             12-24-86

REVISIONS:

DATES:
--------------------------------------------------------------------------

ROUTINE:          tancalc

PURPOSE:          routine used to calculate the tangent at the break points
                  of curve segments in the cross sections

CALLING PROCEDURE/
  DECLARATIONS:
                  tancalc (a,b,&dcos,&dsin)

                  float a[4],b[4],*dcos,*dsin;

INPUT VARIABLES:
                  a,b - [4], differences of x,y curve values about the
                       break point

OUTPUT VARIABLES:
                  dcos,dsin - x,y components of the tangent vector

EXTERNAL VARIABLES:      none

GLOBAL VARIABLES:        none

ROUTINES INVOKED:
                  sqr
                  sqrt
                  fabs

AUTHOR:           James Schwing, Old Dominion University

DATE:             12-24-86

REVISIONS:

DATES:
--------------------------------------------------------------------------

ROUTINE:          sep_seg

PURPOSE:          routine used to separate all information concerning
                  an indicated curve segment in the appropriate arrays
                  for minimum error processing

CALLING PROCEDURE/
```

```
  DECLARATIONS:
      sep_seg(xp,yp,dvect_in,dvect_out,brkpt,segno,ncs,x,y,d0,d1)

      float xp[][MAXCS],yp[][MAXCS],dvect_in[][MAXCS][2],dvect_out[][MAXCS][2]
            *x[][MAXCS],*y[][MAXCS],d0[][2],d1[][2];
      int   brkpt[][MAXCS],segno,ncs;

INPUT VARIABLES:
                xp,yp,dvect_in,dvect_out,brkpt,segno,ncs - as described
                     in "gen_bz"

OUTPUT VARIABLES:
                x,y,d0,d1 - as described in "gen_bz"

EXTERNAL VARIABLES:     none

GLOBAL VARIABLES:       none

INTERNAL VARIABLES:
                cs,i - counters
                istart,iend - markers for the start/end subscripts of
                     the current curve segment in xp,yp

ROUTINES INVOKED:       none

AUTHOR:         James Schwing, Old Dominion University

DATE:           12-24-86

REVISIONS:

DATES:

----------------------------------------------------------------------

ROUTINE:        solve1

PURPOSE:        routine used to solve the least squares equations without
                constraint for the values representing the length of the
                Bezier control points from their respective end points

CALLING PROCEDURE/
  DECLARATIONS:
        solve1(a,b,c,d,f,ncs,n,x,y,lambda)

        float a[],b[],c[],d[],f[],x[][MAXCS],y[][MAXCS],*lambda[][2];
        int   ncs,n[][MAXCS];

INPUT VARIABLES:
                a,b,c,d,f - as described in "bz_minerr1"
                ncs,n,x,y - as described in "gen_bz"
```

```
OUTPUT VARIABLES:
                lambda - as described in "gen_bz"

EXTERNAL VARIABLES:     none

GLOBAL VARIABLES:       none

INTERNAL VARIABLES:
                cs - counter
                det - partial determinant of the least squares matrix
                v - temporary vector

ROUTINES INVOKED:
                sqr
                ln_vect

AUTHOR:         James Schwing, Old Dominion University

DATE:           12-24-86

REVISIONS:

DATES:

---------------------------------------------------------------------

ROUTINE:        solve2

PRUPOSE:        routine used to solve the least squares equations
                constrained so that "lambda[0] / lambda_old" is
                constant over all cross sections; first solving for
                "r" the appropriate value of that ratio then "lambda"
                the values representing the length of the Bezier control
                points from their respective end points

CALLING PROCEDURE/
  DECLARATIONS:
                solve2(a,b,c,d,f,ncs,segno,n,lambda_old,x,y,lambda)

                float a[],b[],c[],d[],f[],lambda_old[],x[][MAXCS],y[][MAXCS],
                      *lambda[][2];
                int   ncs,segno,n[][MAXCS];

INPUT VARIABLES:
                a,b,c,d,f - as described in "bz_minerr1"
                ncs,segno,n,lambda_old,x,y - as described in "gen_bz"

OUTPUT VARIABLES:
                lambda - as described in "gen_bz"

EXTERNAL VARIABLES:     none
```

```
GLOBAL VARIABLES:       none

INTERNAL VARIABLES:
                tot1,tot2 - accumulators
                r - ratio described above
                cs,segpl - counters
                v - temproary vector

ROUTINES INVOKED:
                ln_vect

AUTHOR:         James Schwing, Old Dominion University

DATE:           12-24-86

REVISIONS:

DATES:

----------------------------------------------------------------------

ROUTINE:        get_cpts

PURPOSE:        routine which converts the "lambda" lengths of Bezier
                control points to coordinates for a given curve
                segment over all cross sections

CALLING PROCEDURE/
  DECLARATIONS:
                get_cpts(xp,yp,zp,d0,d1,brkpt,segno,ncs,lambda,cpts)

                float xp[][MAXCS],yp[][MAXCS],zp[],d0[][2],d1[][2],lambda[][2]
                      *cpts[][MAXCS][4][3];
                int   brkpt[][MAXCS],segno,ncs;

INPUT VARIABLES:
                xp,yp,d0,d1,brkpt,segno,ncs,lambda - as described in
                     "gen_bz"
OUTPUT VARIABLES:
                cpts - as described in "gen_bz"

EXTERNAL VARIABLES:     none

GLOBAL VARIABLES:       none

INTERNAL VARIABLES:
                cs,i - counters
                istart,iend - indecies for the start/end points of a
                     given curve segment

ROUTINES INVOKED:       none
```

AUTHOR: James Schwing, Old Dominion University

DATE: 12-24-86

REVISIONS:

DATES:

```
/*
   ROUTINES USED IN THE GENERATION OF MINIMUM ERROR BEZIER
   CURVES CORRESPONDING TO A SET OF CROSS SECTIONS.

           46     gen_bz
           95     seg_info
          145     get_dvect
          319     bz_minerrl
          424     ln_vect
          440     setscl
          460     extrap
          476     tancalc
          511     sep_seg
          551     solve1
          584     solve2
          639     get_cpts                                          */

/* ------------------------------------------------------------ */

#include <sgimath.h>
#include <gl.h>
#include "../include.dir/act_data.h"

#define sqr(X) (X) * (X)
#define MAXCPCSP1 11
#define MAXCPCSP2 12

/* ------------------------------------------------------------ */

seg_info();
get_dvect();
sep_seg();
bz_minerrl();
get_cpts();
solve1();
solve2();
float extrap();
tancalc();
setscl();
float ln_vect();

/* ------------------------------------------------------------ */

gen_bz(xp,yp,zp,ifl,ncs,npcs,cpts,err)

float xp[][MAXCS],yp[][MAXCS],zp[],cpts[][MAXCS][4][3];
int   ifl[][MAXCS],ncs,npcs[],*err;

{ int   n[MAXCPCSP1][MAXCS],nseg,brkpt[MAXCPCSP1][MAXCS],cs,segno;
  float dvect_in[MAXCPCSP1][MAXCS][2],dvect_out[MAXCPCSP1][MAXCS][2],
```

```
        x[MAXPPCS][MAXCS],y[MAXPPCS][MAXCS],d0[MAXCS][2],d1[MAXCS][2],
        lambda_old[MAXCS],lambda[MAXCS][2];

                                    /* separate basic curve information */
   seg_info(ifl,ncs,npcs,n,&nseg,brkpt,err);

   if (err != 0) {
      printf("Inconsistant data; Bezier calculation terminated\n");
      return(*err);
   }

                                    /* get tangent vectors for ALL curve */
                                    /* segment end points                */
   get_dvect(xp,yp,ncs,nseg,n,ifl,brkpt,dvect_in,dvect_out);

   for (cs = 0 ; cs < ncs ; cs++)
      lambda_old[cs] = 1;


                                    /* loop over each curve segment */
   for (segno = 0 ; segno < nseg ; segno++) {

                                    /* separate the curve segment info */
      sep_seg(xp,yp,dvect_in,dvect_out,brkpt,segno,ncs,x,y,d0,d1);

                                    /* minimize the approximation error */
      bz_minerr1(x,y,d0,d1,ncs,n,segno,lambda_old,lambda);

      for (cs = 0 ; cs < ncs ; cs++)
         lambda_old[cs] = lambda[cs][1];

                                    /* save the new control points */
      get_cpts(xp,yp,zp,d0,d1,brkpt,segno,ncs,lambda,cpts);

   }             /* end loop over curve segments */

}


/* -------------------------------------------------------------- */

seg_info(ifl,ncs,npcs,n,nseg,brkpt,err)

int ifl[][MAXCS],ncs,npcs[],n[][MAXCS],*nseg,brkpt[][MAXCS],*err;

{  int cs,seg,segcnt,i,j;

   *err = 0;

                                          /* loop over all cross sections */
   for (cs = 0 ; cs < ncs ; cs++) {
```

```
      seg = 0;
      j = 0;
                                                   /* loop over all point of a given */
                                                   /* cross section                  */
      for (i = 0 ; i < npcs[cs] ; i++) {
                                                   /* a break point between curve     */
                                                   /* segments is ID'd; save that info */
         if (ifl[i][cs] != 0) {
            brkpt[seg][cs] = i;
            n[seg][cs] = j;
            seg++;
            j = 2;
         }
         else                                      /* not a segment end point */
            j++;
      }
      n[seg][cs] = 0;
      segcnt = seg - 1;

      if (cs == 0)                                 /* insure that each cross section  */
         *nseg = segcnt;                           /* has the same number of segments */
      else if (*nseg != segcnt) {
         *err = -1;
         printf("* ERROR: cross section #%d has a different number of curve\n",
                          cs);
         printf("         segments than cross section #1\n");

      } /* end loop over points */

   }   /* end loop over cross sections */

}


/* ------------------------------------------------------------------ */

get_dvect(xp,yp,ncs,nseg,n,ifl,brkpt,dvect_in,dvect_out)

float xp[][MAXCS],yp[][MAXCS],dvect_in[][MAXCS][2],dvect_out[][MAXCS][2];
int   ncs,nseg,n[][MAXCS],ifl[][MAXCS],brkpt[][MAXCS];

{

   float a[4],b[4],ct,st,v[2],len,xel,xe2,yel,ye2;
   int   cs,i,seg,nl,n2,ibase;
```

```
                                        /* loop over all cross sections */
  for (cs = 0 ; cs < ncs ; cs++) {

                                        /* loop over all segment end points
     for (seg = 0 ; seg <= nseg ; seg++) {

        n1 = n[seg][cs];                /* # of points in prior segment
        n2 = n[seg + 1][cs];            /* # of points in following segment
        ibase = brkpt[seg][cs];         /* subscript of the breakpoint

                                        /* CASE: continuous derivative */
                                        /*   neither curve linear      */
        if ((n1 > 2) && (n2 > 2) && (ifl[ibase][cs] == 1)) {

           a[0] = xp[ibase - 1][cs] - xp[ibase - 2][cs];
           a[1] = xp[ibase][cs] - xp[ibase - 1][cs];
           a[2] = xp[ibase + 1][cs] - xp[ibase][cs];
           a[3] = xp[ibase + 2][cs] - xp[ibase + 1][cs];
           b[0] = yp[ibase - 1][cs] - yp[ibase - 2][cs];
           b[1] = yp[ibase][cs] - yp[ibase - 1][cs];
           b[2] = yp[ibase + 1][cs] - yp[ibase][cs];
           b[3] = yp[ibase + 2][cs] - yp[ibase + 1][cs];

           tancalc(a,b,&ct,&st);

           dvect_in[seg][cs][0] = -ct;
           dvect_in[seg][cs][1] = -st;
           dvect_out[seg][cs][0] = ct;
           dvect_out[seg][cs][1] = st;
         }

                                        /* CASE: continuous derivative */
                                        /*   prior curve linear        */
        else if ((n1 == 2) && (ifl[ibase][cs] == 1)) {

           v[0] = xp[ibase - 1][cs] - xp[ibase][cs];
           v[1] = yp[ibase - 1][cs] - yp[ibase][cs];
           len = ln_vect(v);
           ct = v[0] / len;
           st = v[1] / len;
           dvect_in[seg][cs][0] = ct;
           dvect_in[seg][cs][1] = st;
           dvect_out[seg][cs][0] = -ct;
           dvect_out[seg][cs][1] = -st;
         }

                                        /* CASE: continuous derivative */
                                        /*   following curve linear    */
        else if ((n2 == 2) && (ifl[ibase][cs] == 1)) {

           v[0] = xp[ibase + 1][cs] - xp[ibase][cs];
```

```
            v[1] = yp[ibase + 1][cs] - yp[ibase][cs];
            len = ln_vect(v);
            ct = v[0] / len;
            st = v[1] / len;
            dvect_in[seg][cs][0] = -ct;
            dvect_in[seg][cs][1] = -st;
            dvect_out[seg][cs][0] = ct;
            dvect_out[seg][cs][1] = st;
          }
                                        /* all remaining CASES have */
                                        /* discontinuous derivative */
        else {

                                        /* incoming derivative calculation */
                                        /*   prior curve linear            */
            if (n1 == 2) {

               v[0] = xp[ibase - 1][cs] - xp[ibase][cs];
               v[1] = yp[ibase - 1][cs] - yp[ibase][cs];
               len = ln_vect(v);
               dvect_in[seg][cs][0] = v[0] / len;
               dvect_in[seg][cs][1] = v[1] / len;
             }

                                        /* incomming derivative calculation */
                                        /*   prior curve non-linear         */
            else if (seg != 0) {

               xe1 = extrap(xp[ibase][cs],xp[ibase - 1][cs],xp[ibase - 2][cs])
               xe2 = extrap(xe1,xp[ibase][cs],xp[ibase - 1][cs]);
               ye1 = extrap(yp[ibase][cs],yp[ibase - 1][cs],yp[ibase - 2][cs])
               ye2 = extrap(ye1,yp[ibase][cs],yp[ibase - 1][cs]);
               a[0] = xp[ibase - 1][cs] - xp[ibase - 2][cs];
               a[1] = xp[ibase][cs] - xp[ibase - 1][cs];
               a[2] = xe1 - xp[ibase][cs];
               a[3] = xe2 - xe1;
               b[0] = yp[ibase - 1][cs] - yp[ibase - 2][cs];
               b[1] = yp[ibase][cs] - yp[ibase - 1][cs];
               b[2] = ye1 - yp[ibase][cs];
               b[3] = ye2 - ye1;

               tancalc(a,b,&ct,&st);

               dvect_in[seg][cs][0] = -ct;
               dvect_in[seg][cs][1] = -st;
             }
                                        /* outgoing derivative calculation */
                                        /*   following curve linear        */
            if (n2 == 2) {
```

```
                v[0] = xp[ibase + 1][cs] - xp[ibase][cs];
                v[1] = yp[ibase + 1][cs] - yp[ibase][cs];
                len = ln_vect(v);
                dvect_out[seg][cs][0] = v[0] / len;
                dvect_out[seg][cs][1] = v[1] / len;
              }

                                            /* outgoing derivative calculation *
                                            /*   following curve non-linear    *
             else if (seg != nseg) {

                xe1 = extrap(xp[ibase][cs],xp[ibase+1][cs],xp[ibase+2][cs]);
                xe2 = extrap(xe1,xp[ibase][cs],xp[ibase+1][cs]);
                ye1 = extrap(yp[ibase][cs],yp[ibase+1][cs],yp[ibase+2][cs]);
                ye2 = extrap(ye1,yp[ibase][cs],yp[ibase+1][cs]);
                a[0] = xe1 - xe2;
                a[1] = xp[ibase][cs] - xe1;
                a[2] = xp[ibase + 1][cs] - xp[ibase][cs];
                a[3] = xp[ibase + 2][cs] - xp[ibase + 1][cs];
                b[0] = ye1 - ye2;
                b[1] = yp[ibase][cs] - ye1;
                b[2] = yp[ibase + 1][cs] - yp[ibase][cs];
                b[3] = yp[ibase + 2][cs] - yp[ibase + 1][cs];

                tancalc(a,b,&ct,&st);

                dvect_out[seg][cs][0] = ct;
                dvect_out[seg][cs][1] = st;
             }

          }

       }  /* end loop over break points */


                                            /* reset the first and last      */
                                            /* derivative values as required */
       ct = dvect_out[0][cs][0];
       st = dvect_out[0][cs][1];
       setscl(&ct,&st);
       dvect_out[0][cs][0] = ct;
       dvect_out[0][cs][1] = st;

       ct = dvect_in[nseg][cs][0];
       st = dvect_in[nseg][cs][1];
       setscl(&ct,&st);
       dvect_in[nseg][cs][0] = ct;
       dvect_in[nseg][cs][1] = st;

    }  /* end loop over cross sections */
```

```
}


/* ------------------------------------------------------------------ */

bz_minerr1(x,y,d0,d1,ncs,n,segno,lambda_old,lambda)

float x[][MAXCS],y[][MAXCS],d0[][2],d1[][2],lambda_old[],lambda[][2];
int   ncs,n[][MAXCS],segno;

{  float dist,s[MAXCS][5],scomp[MAXCS][5],sx2m3[MAXCS],sx2p1[MAXCS],l1,l2,l3
         r1,r2,r3,r4,r5,r6,r,tot1,tot2,totald,
         a[MAXCS],b[MAXCS],c[MAXCS],d[MAXCS],f[MAXCS];
   int   i,cs,p,n_cs,nm1,nm2;


   for (cs = 0 ; cs < ncs ; cs++) {      /* loop over all cross sections */
      totald = 0;
      n_cs = n[segno + 1][cs];
      nm1 = n_cs -1;
      nm2 = nm1 - 1;


      switch (n_cs) {                    /* number of points in the cross */
                                         /* section                       */

         case 2:                         /* straight line => no error     */
            break;

         default:                        /* error possible in all other cases
                                         /* find chord lengths of data    */
                                         /* positions & related functions */
                                         /* required for matrix set up    */
            for (i = 0 ; i < nm1 ; i++) {
              dist = sqrt(sqr(x[i+1][cs] - x[i][cs])
                          + sqr(y[i+1][cs] - y[i][cs]));
              totald += dist;
              s[i][0] = totald;
            }

            for (i = 0 ; i < nm2 ; i++) {
              s[i][0] /= totald;
              scomp[i][0] = 1 - s[i][0];
              sx2m3[i] = 3 - 2 * s[i][0];
              sx2p1[i] = 1 + 2 * s[i][0];

              for (p = 1 ; p < 4 ; p++) {
                s[i][p] = s[i][p - 1] * s[i][0];
                scomp[i][p] = scomp[i][p-1] * scomp[i][0];
              }
```

```
          }

                                          /* set up the least squares matrix */
                                          /* with information from this      */
                                          /* cross section                   */
          l1 = 0;
          l2 = 0;
          l3 = 0;
          r1 = 0;
          r2 = 0;
          r3 = 0;
          r4 = 0;
          r5 = 0;
          r6 = 0;

          for (i = 0 ; i < nm2 ; i++) {
            l1 += scomp[i][3] * s[i][1];
            l2 += scomp[i][2] * s[i][2];
            l3 += scomp[i][1] * s[i][3];
            r1 += scomp[i][1] * s[i][0] * (x[i + 1][cs] * d0[cs][0]
                                + y[i + 1][cs] * d0[cs][1]);
            r2 += scomp[i][3] * s[i][0] * sx2p1[i];
            r3 += scomp[i][1] * s[i][2] * sx2m3[i];
            r4 += scomp[i][0] * s[i][1] * (x[i + 1][cs] * d1[cs][0]
                                + y[i + 1][cs] * d1[cs][1]);
            r5 += scomp[i][2] * s[i][1] * sx2p1[i];
            r6 += scomp[i][0] * s[i][3] * sx2m3[i];
          }

          a[cs] = 3 * l2 * (d0[cs][0] * d1[cs][0] + d0[cs][1] * d1[cs][1])
                      * lambda_old[cs];
          b[cs] = 3 * l3 * (sqr(d1[cs][0]) + sqr(d1[cs][1]));
          c[cs] = r4 - r5 * (x[0][cs] * d1[cs][0] + y[0][cs] * d1[cs][1])
                    - r6 * (x[nm1][cs] * d1[cs][0] + y[nm1][cs] * d1[cs][1]);
          d[cs] = 3 * l1 * (sqr(d0[cs][0]) + sqr(d0[cs][1]))
                      * sqr(lambda_old[cs]);
          f[cs] = (r1 - r2 * (x[0][cs] * d0[cs][0] + y[0][cs] * d0[cs][1])
                    - r3 * (x[nm1][cs] * d0[cs][0] + y[nm1][cs] * d0[cs][1]))
                      * lambda_old[cs];
          break;
        }

    }  /* end loop over cross sections */

                                          /* solve the first curve segment */
    if (segno == 0)                       /* no prior "r" constraint       */
      solve1(a,b,c,d,f,ncs,n,x,y,lambda);
                                          /* solve other curve segments     */
    else                                  /* use common "r" at each start point */
       solve2(a,b,c,d,f,ncs,segno,n,lambda_old,x,y,lambda);
```

```
}


/* ---------------------------------------------------------------- */

float ln_vect(v)

float v[2];

{   float len;

    len = sqrt(sqr(v[0]) + sqr(v[1]));

    return(len);

}


/* ---------------------------------------------------------------- */

setscl(ct,st)

float *ct,*st;

{

    if (fabs(*ct) < .05) {
        *ct = 0;
        *st = 1;
    }
    else if (fabs(*st) < .05) {
        *ct = 1;
        *st = 0;
    }

}

/* ---------------------------------------------------------------- */

float extrap(a,b,c)

float a,b,c;

{   float d;

    d = 3 * (a - b) + c;

    return d;
```

```
}


/* ------------------------------------------------------------- */

tancalc (a,b,dcos,dsin)

float a[4],b[4],*dcos,*dsin;

{  float a0,b0,t1,t2,w2,w3;

   w2 = fabs(a[2] * b[3] - a[3] * b[2]);
   w3 = fabs(a[0] * b[1] - a[1] * b[0]);

   a0 = w2 * a[1] + w3 * a[2];
   b0 = w2 * b[1] + w3 * b[2];

   t1 = sqrt(a0 * a0 + b0 * b0);

                        /* If the curve is a straight line, then     */
                        /*  b[i]/a[i] = b[i+1]/a[i+1] = tan(a)       */
                        /*  for all i = 0,1,2                        */
                        /*  and thus a0 = b0 = 0                     */
                        /* treat this case separately.               */
   if ((a0 == 0) && (b0 == 0)) {
      t2 = sqrt(sqr(a[0]) + sqr(b[0]));
      *dcos = a[0] / t2;
      *dsin = b[0] / t2;
    }
   else {
      *dcos = a0 / t1;
      *dsin = b0 / t1;
   }

}


/* ------------------------------------------------------------- */

sep_seg(xp,yp,dvect_in,dvect_out,brkpt,segno,ncs,x,y,d0,d1)

float xp[][MAXCS],yp[][MAXCS],dvect_in[][MAXCS][2],dvect_out[][MAXCS][2],
      x[][MAXCS],y[][MAXCS],d0[][2],d1[][2];
int   brkpt[][MAXCS],segno,ncs;

{  int cs,istart,iend,i,j;

                              /* loop over all cross sections */
   for (cs = 0 ; cs < ncs ; cs++) {
```

```
                                            /* subscripts for the start and end */
                                            /* of the current curve segment in  */
                                            /* this cross section               */
        istart = brkpt[segno][cs];
        iend = brkpt[segno + 1][cs];
        j = 0;

                                            /* loop over points of this segment */
                                            /* copy to "x" & "y"                */
        for (i = istart ; i <= iend ; i++) {

           x[j][cs] = xp[i][cs];
           y[j][cs] = yp[i][cs];
           j++;
        } /* end loop over points */

                                            /* copy derivatives for this segment */
        d0[cs][0] = dvect_out[segno][cs][0];
        d0[cs][1] = dvect_out[segno][cs][1];
        d1[cs][0] = dvect_in[segno + 1][cs][0];
        d1[cs][1] = dvect_in[segno + 1][cs][1];

     }  /* end loop ove cross sections */

  }


  /* -------------------------------------------------------------------- */

  solvel(a,b,c,d,f,ncs,n,x,y,lambda)

  float a[],b[],c[],d[],f[],x[][MAXCS],y[][MAXCS],lambda[][2];
  int   ncs,n[][MAXCS];

  {  int   cs;
     float det,v[2];

                                            /* loop over cross sections */
     for (cs = 0 ; cs < ncs ; cs++) {

                                            /* CASE: non-linear curve segment */
                                            /*   solve the matrix             */
        if (n[1][cs] != 2) {
           det = d[cs] * b[cs] - sqr(a[cs]);
           lambda[cs][0] = (f[cs] * b[cs] - c[cs] * a[cs]) / det;
           lambda[cs][1] = (c[cs] * d[cs] - f[cs] * a[cs]) / det;
        }
                                            /* CASE: linear segment */
        else {
           v[0] = x[1][cs] - x[0][cs];
           v[1] = y[1][cs] - y[0][cs];
```

```
            lambda[cs][0] = ln_vect(v) / 3.;
            lambda[cs][1] = lambda[cs][0];
        }

    }  /* end loop over cross sections */

}


/* ---------------------------------------------------------------- */

solve2(a,b,c,d,f,ncs,segno,n,lambda_old,x,y,lambda)

float a[],b[],c[],d[],f[],lambda_old[],x[][MAXCS],y[][MAXCS],lambda[][2];
int   ncs,segno,n[][MAXCS];

{   float tot1,tot2,r,v[2];
    int   cs,segp1;

    tot1 = 0;
    tot2 = 0;
    segp1 = segno + 1;
                                /* solve first for the required common */
                                /* ratio                               */

                                /* loop over all cross sections */
    for (cs = 0 ; cs < ncs ; cs++) {

        if (n[segp1][cs] != 2) {
           tot1 += f[cs] - (a[cs] *c[cs]) / b[cs];
           tot2 += d[cs] - sqr(a[cs]) / b[cs];
        }
    }  /* end loop over cross sections */

    r = tot1 / tot2;

                                /* use "r" to generate the first control */
                                /* point, then solve for the other       */

                                /* loop over all cross sections          */
    for (cs = 0 ; cs < ncs ; cs++) {

                                /* first control point */
        lambda[cs][0] = r * lambda_old[cs];

                                /* second control point     */
                                /* CASE: non-linear segment */
        if (n[segp1][cs] != 2)
           lambda[cs][1] = (c[cs] - (a[cs] * r)) / b[cs];

                                /* CASE: linear segment     */
```

```
      else {
         v[0]  = x[1][cs] - x[0][cs];
         v[1]  = y[1][cs] - y[0][cs];
         lambda[cs][1] = ln_vect(v) / 3.;
      }

   }  /* end loop over cross sections */

}


/* -------------------------------------------------------------- */

get_cpts(xp,yp,zp,d0,d1,brkpt,segno,ncs,lambda,cpts)

float xp[][MAXCS],yp[][MAXCS],zp[],d0[][2],d1[][2],lambda[][2],
      cpts[][MAXCS][4][3];
int   brkpt[][MAXCS],segno,ncs;

{  int cs,istart,iend,i;

                              /* store the control points of this */
                              /* segment for each cross section   */

                              /* loop over each cross section */
   for (cs = 0 ; cs < ncs ; cs++) {

      istart = brkpt[segno][cs];
      iend = brkpt[segno + 1][cs];

      cpts[segno][cs][0][0] = xp[istart][cs];
      cpts[segno][cs][0][1] = yp[istart][cs];
      cpts[segno][cs][1][0] = xp[istart][cs] + lambda[cs][0] * d0[cs][0];
      cpts[segno][cs][1][1] = yp[istart][cs] + lambda[cs][0] * d0[cs][1];
      cpts[segno][cs][2][0] = xp[iend][cs] + lambda[cs][1] * d1[cs][0];
      cpts[segno][cs][2][1] = yp[iend][cs] + lambda[cs][1] * d1[cs][1];
      cpts[segno][cs][3][0] = xp[iend][cs];
      cpts[segno][cs][3][1] = yp[iend][cs];

      for (i = 0 ; i < 4 ; i++)
         cpts[segno][cs][i][2] = zp[cs];
   }

}  /* end loop over cross sections */
```